ICASE

COMMENTS ON HIGHLY RELIABLE

SOFTWARE FOR AVIONICS APPLICATIONS

Jacob T. Schwartz

Report No. 81-31

September 23, 1981

INSTITUTE FOR COMPUTER APPLICATIONS IN SCIENCE AND ENGINEERING

NASA Langley Research Center, Hampton, Virginia 23665

Operated by the

UNIVERSITIES SPACE USRA RESEARCH ASSOCIATION

COMMENTS ON HIGHLY RELIABLE SOFTWARE FOR AVIONICS APPLICATIONS

Jacob T. Schwartz
Courant Institute of Mathematical Sciences
New York University

ABSTRACT

The differences between hardware and software reliability for digital systems are discussed in the context of applications where a failure may result in the loss of human life. In particular, it is argued that techniques for guaranteeing reliability of hardware are not necessarily appropriate for software. The potential of a variety of approaches for assuring software reliability is discussed.

This research was supported under NASA Contracts No. NAS1-14472 and NAS1-15810 while the author was in residence at the Institute for Computer Applications in Science and Engineering, NASA Langley Research Center, Hampton, VA 23665.

1. The fundamental difference between hardware and software reliability considerations

To be willing to apply computer technology fully to avionics or other critical applications, one would have to be convinced that the computer systems supplied, including both hardware and software, were more reliable than the extensively tested manual/hydraulic/electronic systems which the computer systems were to replace. To attain and defend this conviction will not be easy. Although quite challenging, the purely hardware aspect of this problem is probably manageable by appropriately adapting present hardware reliability determination techniques to complex digital electronics. Software, however, is another matter, in part because we lack even an adequate conceptual framework for assessing software reliability in any quantitative way. It is therefore not at all clear how to proceed in trying to certify the 'flight-worthiness' of any given software module or system. Nevertheless, it is obvious that pressures for such certification will continue to grow. Leaping in where angels fear to fly, this informal note will attempt to make a few points which may help clarify some of the very vexing problems that will inevitably confront certification agencies as they struggle pragmatically with the software enigma.

The first assertion I would like to make is that software reliability is a matter qualitatively different from hardware reliability. Consequently attempts, of which there have been many, to apply concepts drawn from hardware reliability theory to software tend to mask rather than clarify the problems that must be faced in trying to assure the reliability of software systems. Hardware reliability theory assumes that the hardware systems which it considers malfunction and decay stochastically. Among

other things, this leads to a notion of (preventive) maintenance: by predicting a system's pattern of decay toward failure, and by arresting this decay in its earliest stages, one can aim to keep a system operating close to its normal condition for extended periods of time. The decay assumed is in effect a process of stochastic wear.

None of these concepts apply to software. This is shown by the fact that all computer centers schedule periodic preventative maintenance of their hardware systems, but that none would dream of scheduling software systems for maintenance which was either periodic or 'preventative'. Indeed, if one knew what tests to apply during such 'preventative' maintenance, one would apply them before initial system release; once these tests were passed, they would never have to be applied again unless the software changed.

A simple thought-experiment serves to emphasize this point. Suppose I am about to board a (computer controlled) aircraft. If I am told that its hardware has just been checked over by a skilled mechanic, who has just replaced every slightly defective part, I will board it with increased confidence. However, if I am told that its software has just been checked over by a skilled programmer, who has just replaced every slightly defective subprocedure, my desire not to board for a considerable period of time will be overwhelming.

2. 'Design faults' and 'discontinuous behavior' in software reliability

Part of the reason for the fundamental difference in attitude revealed by the preceding reflection lies in the fact that software failures are not consequences of wear; rather, they are encounters with design faults. (Here 'design' is used in a sense including 'implementation'.) Another thought-

experiment can help clarify this point. A stochastic reliability model of the ordinary sort might justify the statement that the probability of an aircraft suddenly failing in flight is less than 10^{-9}. It is, however, not hard to undercut the confidence which such an estimate seems to justify. To do so, we have only to transpose, into the hardware realm, the type of failure characteristic of software. Suppose, for example, that I assert of the commercial airliner CD744 that if simultaneously

(a) Its flaps and rudder are in a particular condition;

(b) Its airspeed is x and the turning radius is R;

(c) The ambient air pressure is exactly p;

then its aerodynamic design is such that it will go into an unrecoverable stall. (It is assumed here that these hypothetical failure parameters are narrowly defined, but that they lie somewhere in the range of normal commercial operation.) Suppose that I assert in addition that the only reason that this catastrophic failure mode has not been observed is that the listed parameters never simultaneously took on these critical values, either in the hundreds of hours of precertification testing to which the aircraft was originally subjected, or during the tens of thousands of hours of commercial use that it has logged subsequently. If this assertion is believed plausible, then the overwhelming part of the probability that the CD744 should fail suddenly in flight coincides with the probability that it should wander into the 'design fault' parameter region described above. Whatever number one cares to ascribe to this probability, it is clearly much larger than (and also qualitatively different from) the hypothetical 10^{-9} which a standard stochastic failure model might yield.

Thus our willingness to regard the number produced by a stochastic failure model as a quantity that is at all significant rests on an unspoken major premise: that hardware simply does not fail in this discontinuous way, i.e. that the behavior of hardware does not depend upon its control parameters in a manner characterized by sudden, sharp breaks. Put differently, we expect the response of physical systems to their control parameters to be monotone increasing in some ranges, monotone decreasing in others. Therefore we believe that, by testing the response at a limited number of points, we can at least bound their response at untested points. This rules out the possibility of unsuspected narrow 'pitfalls'. But it is precisely on encountering such pitfalls that software fails.

3. Software failures

Thus the rate at which software fails is not at all the consequence of a stochastic wear process. Rather, it reflects the rate at which design pitfalls present in the software from its inception are encountered as execution traverses the state space of a given program. This rate depends on the manner in which execution wanders through the logical space S of states of the program.

To visualize this situation, one can think of S as a k-parameter space of some kind, and think of the faults in it as stones scattered through the body of a cake. As execution proceeds it will traverse S in a manner that is partly random, and may occasionally collide with a fault. When this happens, a very carefully controlled fix may succeed in removing the fault without introducing any new errors. Repeated fixes of this carefully controlled kind can progressively reduce the density of the faults remaining in the region of the logical space S which the

usage generated by a particular application regularly traverses. On the other hand, faults in untraversed program regions will simply remain, and will only reveal themselves when usage shifts into some new region of the space S . Finally, any change made in a program P from which faults have been removed by subjecting P to extensive varied usage and repeated repair will re-introduce a new scattering of faults which will be as difficult to remove as faults present from the beginning. Changes which are uncontrolled or which have global logical effects are particularly likely to have this degrading effect.

These remarks correspond to two common observations concerning software errors:

(i) Errors are particularly likely to occur just after a program has been modified.

(ii) Whenever the pattern in which a program is used shifts significantly, design faults that have remained latent within it (perhaps for extended periods) are likely to reveal themselves in particularly large numbers.

Normal software test and maintenance procedures reflect some of these observations. Careful testing will always try to subject a program to systematically varied use, e.g. by insisting that every line of code must be executed by the tests, and that the test must force all conditional branches to be taken in both of their two possible directions. Test libraries are generally preserved so that they can be reapplied as 'regression' tests whenever a code has been modified in any way. Finally, the experienced programmer will never remove any test code that has proved useful from a program's source text, since he knows that it may become necessary to revert to development-phase diagnostic activities whenever

a shift in usage pattern has revealed the presence of some hitherto unsuspected design fault.

Note that this conceptual model of the process of software failure differs in several very essential regards from the normal model of hardware failure. In no sense do we think of program usage as introducing faults into a program by any statistical process of wear or decay; thus program use in a fixed or relatively fixed pattern does not test it in the same way that protracted use would test hardware. Finally, our model leaves no room for any notion of 'preventative' software maintenance.

4. Detecting and eliminating discontinuous failures in complex designs

Our limited ability to deal successfully with errors in software systems was seen in the preceding section to trace to the inherently discontinuous nature of these errors. To fight against these limits one must

(a) Reduce the initial likelihood of such errors creeping in;

(b) Improve the effectiveness of methods for detecting hidden logical discontinuities;

(c) 'Smooth out' these logical faults, if some way to do this can be found;

(d) Stabilize software as much as possible, to ensure that new faults do not creep into a system from which prior faults have been slowly and expensively removed.

I will now comment on various approaches which these remarks suggest.

(a) Preventing logical errors

Typical errors in software systems arrange themselves along a spectrum from superficial 'blunders' to deep-lying misapprehensions

concerning the logical/mathematical behavior of a complex algorithm. A typical blunder might be the misspelling of a variable name or the use of the value of a quantity where a pointer to the quantity is really required. Improper treatment of marginal cases, for example empty arrays, full arrays, or internal quantities which have reached their maximum allowed limits, forms an important, and often persistent, intermediate category of errors. Misunderstandings concerning the correct form of an algorithm can be particularly troublesome during program development, but their crippling effects are normally so pervasive that they tend to be corrected before a program comes into operational use.

In certifying software for avionics applications one will be concerned principally with faults capable of surviving a period of careful and systematic testing. Such testing is likely to detect most simple blunders, but some fraction of such blunders can creep through, and one will therefore be concerned to reduce their number as much as possible. The best way of doing this is to subject the text of a program to redundancy requirements upon which programmed static checks can be based. In particular, the use of a strongly-typed language ought to be insisted on. It is even desirable to carry the typing of variables which such languages support even further, e.g. by allowing a programmer to declare more refined logical types than the basic types of the language being used (for example, variables characterized by their physical dimensions), and to declare the pattern in which operations apply to objects of this type, and also the types of results returned. A computer can then examine for these and other types of errors, for example potentially uninitialized variables.

(ii) Blunders creep into code in proportion to the length of the code, and their numbers rise with the degree of technical artifice which the code employs. It is possible to diminish both the length and the artificiality of code by writing initial versions of it in a suitable very high level language, and by transforming this initial version into a final production version via mechanical or semi-mechanical steps whose likelihood of correctness can be kept high. (Another possibility here is to check mechanically for certain types of formal correspondence between a very high level primary code version and a production version which is supposedly equivalent to this primary code. In effect, this uses the very high level version as a tool for securing redundancy along a logical dimension too sophisticated to be reached by more primitive type-declarations.) Later we will see that such a very high level code version can be useful in other ways also, for example in facilitating the work of 'tiger teams' conducting adversary code audits, and also in stabilizing at least some version of a code which may have to be adapted to changing hardware environments.

(iii) Very systematic testing of code, e.g. strict application of the rule that it must all be exercised, that all branches must be taken in both directions, that a variable should receive data at each of its points of use from all its potential points of definition, etc., can probably eliminate the great majority of superficial blunders. This leaves errors of moderate logical depth, e.g. mishandling of marginal cases, as the likeliest source of failure in a code that has been very carefully developed and tested. Very systematic testing should also eliminate many of the errors of this class. On the other hand, a substantial fraction of such errors must be expected to survive even quite thorough testing,

especially if these errors strike only in peculiar combinations of circumstances. Though it is not at all clear how to smoke out errors which are even this deeply concealed, we will try later to suggest a pragmatic technique which might afford some substantial level of protection against catastrophic surprises.

(iv) To rule out errors which are deeper than type errors, failures to initialize or to update, and other equally superficial faults, one needs to penetrate into a program's logic. To do this, formal mathematical tools and methods, for example the apparatus of checkable proof which mathematical logic supplies, would have to be used. We now turn to assess the likelihood that such techniques can be applied successfully.

Formal proofs of program correctness

Techniques for proving programs mathematically correct are known, and after some decades of work are just beginning to result in systems that are more than laboratory playthings. One can, therefore, hope to use such techniques to guarantee the correctness of programs to be used in critical applications. However, in spite of the steadily increasing importance of this approach, it does not appear likely to play more than a secondary role during the next few decades. The following observations support this guardedly pessimistic judgment.

(i) In the literature on formal proofs of program correctness, one finds references to two significantly different levels of proof, which are not always distinguished as carefully as they should be. These are fully machine-checked proofs on the one hand, and detailed manual reasoning, resembling mathematical proofs of the kind ordinarily published, on the other. Proofs of the first kind are still extremely expensive, and have been carried out in only a very few cases. Moreover, there is reason to

believe that, in some of the numerical areas with which avionic software will have to deal, formal proof techniques will be particularly hard to apply. On the other hand, proofs that are not machine-checked but only verified manually do not give any enormously high degree of protection against failure, since incorrect handling of marginal cases, which is a crucial kind of program fault, tends also to be ubiquitous in ordinary mathematical proof. Thus manually verified proof can best be regarded as a semi-formal technique that can help an adversary 'tiger team' (of the kind to be discussed below) to smoke out errors that would otherwise be overlooked. However, if it is to play this role it is essential that the verification formalism used should retain intuitive clarity rather than dissolving into a repellent cloud of obscure formal details. Thus a verification formalism properly designed to assist manual verification would have to have quite a different flavor from one intended as a component of a fully computer-checked system. No existing verification system seems to have been designed with this requirement in mind.

(ii) Programs often fail because of some discrepancy between a clean mathematical notion and the considerably more painful mass of detail used to represent this notion within a computer. Failures are, for example, often associated with overflows of data fields, discrepancies between true integer arithmetic and its approximate (one's or two's complement) computer representation, etc. While not impossible, it would be particularly tedious to bring material of this sort into a formal framework of mathematical proof.

(iii) This last problem becomes particularly acute in the case of real arithmetic, in which connection there arises a yet more fundamental problem. In most cases precise mathematical justification of the approximations used in numerical analysis are simply not known, and hence we are not in position to give formal proofs either of the correctness of these approximations or of systems which rely upon them.

(iv) Overall, it must be admitted that, to make formal proofs of program correctness available on any substantial scale, a challenging technology, of which at present we possess only some basic principles, would have to be put into place. Techniques reducing the presently enormous cost and tedium of checking proofs by machine would have to be developed very greatly. Systems for managing rigidly locked libraries of formally verified program fragments would be required. Practical methods for minimizing the amount of source text that had to be reprocessed every time some part of a software system has been changed would have to be found. We are beginning to understand how to do all these things, and this technology deserves to be patiently yet vigorously cultivated. Even so, it appears unlikely that it will develop enough in the next few years to withstand the shock of substantial practical use. Thus program proving can be regarded as a useful auxiliary tool in justifying the airworthiness of software, but by no means an exclusive, and probably not even an entirely central, tool.

(b) Detecting pitfalls

If not by formal mathematical proof of correctness, then how can potential errors in complex, discontinuous logical systems be detected? Two methods, one of which has already been mentioned, suggest themselves.

These are

(b.i) Subjecting the systems to as wide and as varied use as possible before the system is used in any critical application.

(b.ii) Subjecting the system to close and rigorous examination by groups of skilled adversaries, i.e. 'tiger teams'.

Under (b.i), one's principal aim must be to secure very varied use, rather than extensive but relatively stereotyped use. An ideal would be to use systems or system elements known to be identical in a great variety of very different applications, a few critical, but most non-critical. Given this pattern of usage it would be reasonable to hope that the first appearance of any remaining design fault would be in some non-critical application rather than in a critical one. Then, assuming that error-reporting was rapid and centrally managed, one could hope that enough lead-time would be gained to prevent future errors from affecting any critical situation.

Suppose, to be more specific, that such complex and central systems facilities as process control or interprocessor communications came to be resident on a fixed, mass-produced chip which came into wide use for household and office systems, plant automation, ground transport applications, and so forth. Then it would be highly desirable to use these very same components in avionics applications, as their non-avionics usage would then 'protect' the avionics use against unforeseen fault, in the manner just explained. The key here is rigid stabilization of hardware and software components having both critical and non-critical applications, or, this lacking, rigid stabilization of systems whose long usage in a critical application affords something of this same kind of protection.

'Tiger team' audit of software systems

While by no means exhaustive, an adversary 'tiger team' audit of a software system is probably capable of smoking out numerous errors which would escape detection even by very systematic batteries of tests. Skilled and experienced tiger teams can in principle comprehend the logic of a program deeply enough to surmise the existence of and examine for many faults hidden in the interlocking logical mechanisms of a program.

However, to be effective, such teams would have to operate in a spirit quite different from that of today's ordinary 'design walkthrough'. Such design walkthroughs are normally allotted very limited time and resources, and operate under the assumption that they will approve a software product, if not on a first examination, then at least after the second or the third examination. By contrast, a satisfactory adversary audit would require as much as or more time and manpower than went into system design and coding initially, and might well refuse to pass a system even after repeated examination.

An audit team could refuse to approve a system for various reasons, including

(i) Detected faults (which the audit team could list);

(ii) Lack of fully adequate documentation, or non-transparent design, either of which left the audit team unable to assure itself that the system would work as claimed;

(iii) Insufficient proof that the system supplied is precisely equivalent to that described by its supporting material;

(iv) Inadequate testing.

Note that the possibility of rejection for reason (ii) might act strongly to encourage the development of systems from executable specifications written in very high level languages. This could conduce to design clarity and simplicity. Moreover, if pre-certified very-high-level libraries of important utility functions were available, the problem of obtaining certification for a complex system could be reduced by using these certified techniques, rather than entirely new methods, to implement some of the functions required.

It may be that carefully organized adversary audits will develop as one of the principal methods for certifying the airworthiness of software. This would seem to be an area in which it would be appropriate to mount a few well-chosen experimental efforts. For example:

(i) Two or more independent 'tiger teams' could be assigned to audit the same software systems. The extent to which their findings came to overlap (as the number of audited errors fell) might be taken as an indication of their joint thoroughness of audit.

(ii) A system known to have reached good reliability might be subject, on the one hand to systematic testing in the ordinary sense, on the other hand to adversary audit, and the number and nature of errors detected by each of these two methods compared.

Adversary audit groups may in time come to provide a specialized function analogous to that presently provided by firms of Certified Public Accountants, though of course they would operate in a very different technical area, and in one that was much more critical since loss of life rather than simple financial loss was at stake.

A useful by-product of a successfully completed adversary audit might be a critical exposures list, i.e. a list of the hardware failures from which the software was unable to recover successfully. The availability of such a list might improve the usefulness of hardware reliability assessments very substantially.

Fault-tolerant software

As a possible method for reducing the likelihood that software should fail catastrophically, the following suggestion for 'fault tolerant' software has been advanced.

(i) Find points in a software system at which the validity of an intermediate result R can be checked fairly rapidly.

(ii) Perform this check. If it fails, the code C producing the result R has malfunctioned. In this case, back up the system to exactly the status which it had before the code sequence C was executed. Then execute a different, but logically equivalent sequence C', i.e. one which produces the same result R, but in a different way.

The thought here is that if the 'probability' x that C will fail is small, then the probability that both C and C' will fail is x^2; hence much smaller. The 'backup' necessary to support this scheme can be implemented at a very modest cost in speed, and by multiplying the size of memory required by a small integer factor.

This approach has some arguable advantages, but for the following reasons is not entirely convincing.

(a) The scheme's ability to recover is based upon the ability of C' to generate the acceptable result R that C did not produce. For this to be possible, the common input I to C and C' must be valid; if it is not, then both C and C' will fail, since this failure is caused

by undetected bad data supplied to both of them. When this happens, one can try to back up still further, i.e. to undo the effects of the code block C" executed just before C and try its alternative. But a scheme of this sort rapidly grows complex, and its execution-time cost rises at least quadratically with the number of 'check points' which an error has passed before the occurrence of a logical inconsistency is detected. In a real-time system subject to fairly stringent response-time requirements, this rising execution-time may be intolerable, since it may guarantee that the system will fail whenever it backs up through more than a few levels.

Thus our ability to make software 'fault tolerant' in the sense described depends on our ability to state the logical conditions upon which its functioning depends fully enough to make it unlikely that an error can propagate undetected through several successive checks.

(b) Moreover, the 'fault tolerant' approach is only possible if the check-conditions that need to be verified can be evaluated rapidly enough not to degrade system performance intolerably. Especially if complex tables are being manipulated, and if system functioning depends upon the global consistency of these tables, this may simply not be possible. When such tables are involved it may also be difficult to design an alternative code sequence C' that performs the same logical functions as C but is not identical to C' .

(c) It can also be objected that the 'fault tolerant software' approach does not diminish the probability of system failure to the x^2 level claimed. This low failure probability estimate is only justified if the probabilities that C and C' both fail are independent. To ensure this independence, one can, for example, insist that the codes

C and C' should be developed by two different programming groups, working independently. However, even if this is done, errors in C and in C' may not be independent. For example, both errors may trace back to points at which the common specification from which both C and C' were necessarily produced was ambiguous or incorrect. Moreover, we have seen that errors often trace to mishandled marginal cases or combinations thereof, and the very same touchy case which confused the authors of code C may have defeated the authors of C'.

A final comment on the 'fault tolerant' software approach is that every detected failure of a 'first-line' code sequence C will lead forthwith to the repair of C. After such repair, C will presumably never fail in the same way again. Thus the backup software C' provided for C has the function of protecting against events that happen just once, rather than of protecting against events that, due to wear, will happen infinitely often. This again shows a difference between software and hardware reliability considerations.

(c) 'Smoothing out' logical pitfalls in complex systems

Experience suggests that the points at which much-used software fails are often those points at which some internal table or data structure on which the software depends, but which is only indirectly related to the external physical situation which the software is managing, falls into some internally inconsistent condition. One plausible way of recovering from such situations when they are detected, especially in an avionics application whose sole purpose is to control physical hardware, is simply to terminate execution and to re-initialize completely, restarting all tables from their initial conditions. This would require that the software system must be capable of acquiring all the physical parameters which it

needs directly from measurements that it can make on the physical system being managed, i.e. that all data necessary for control is recoverable ab initio from the aircraft. This seems a reasonable design constraint to impose. If it is accepted, then an attractive range of possibilities opens up.

(i) To 'stress' the software system, it can be forced to re-initialize itself repeatedly, e.g. once each second during periods of experimental flight. This should succeed in exposing the system to many of the conditions which it would have to manage during a real software-fault induced re-initialization, and should make the immediate post-initialization operation of the system particularly reliable. Note that the kind of testing suggested uses the varying physical condition of the aircraft to expose the software to many different test conditions.

(ii) One can run and compare several system versions, identical in their code but initialized at different moments, or one running continuously, the other frequently re-initialized in the manner suggested above. Any failure of these several software processes to transmit substantially the same controls to the aircraft effectuators would point to some potentially dangerous discontinuity.

A system built according to this structural principle might operate as follows. Main-line software would transmit controls to the various physical subsystems aboard an aircraft. Subsidiary 'parasitic' software processes would monitor the activity of this main-line software system, verifying that the control signals it was generating always lay within reasonable ranges, that they kept the aircraft operating in a reasonable range, and that the main-line software was carrying out all its periodic activities on schedule. If any of these checks failed, an immediate

status dump would be taken and saved for later examination; then the system would immediately be re-initialized, would re-acquire the data it needed, and would attempt to continue. Note that re-initialization will always involve re-acquisition of any currently active user-supplied command parameters. These parameters must therefore be held in non-volatile registers. If re-initialization is completely successful, no device control or control display should be lost for more than a fraction of a second; thus the system user need not be actively concerned with the fact that a software fault has interrupted operation, though of course each such incident should be reportable to the software maintenance group and their team of adversaries.

Note that by insisting that avionics software systems be 'fully restartable' in the manner suggested, we are attempting to tie their behavior to the continuous responses of the physical systems which they regulate, thereby 'smoothing out' logical discontinuities which would otherwise allow catastrophic failures unpredictable by testing. Note also that the recovery scheme advanced bears some relationship to the 'fault tolerant' software notion discussed earlier. However, instead of relying on microscopic internal checks of result acceptability, we propose to monitor the software system's external behavior, and instead of trying to apply a restricted correction when a fault is detected we propose to take the cruder and more radical step of re-initializing completely. Moreover, instead of using an alternative code sequence to recover, we use the same code that has just failed, but change the data on which it is acting.

System re-initialization should also involve a series of fast checks to verify that all physical, control, and communication systems are in working order. If some subsystem is found to be inoperable, the software can be programmed to report this failure and allow for operation in a suitable fallback mode. This suggests that ability to check rapidly for the operability of all subsystems may become a design requirement.

5. Stabilization of software systems

The preceding discussion emphasizes the difficulty and high expense that will inevitably attach to attempts to guarantee the correct functioning of critical software. It follows that it will be important to stabilize this software when extensive usage and careful audit have brought it to a condition of high reliability. Here stabilization must mean precise logical stabilization, since any uncontrolled change, even a 'small' one, is apt to reproduce the scattering of small hidden errors in which the danger of failure lies.

The following techniques might help to achieve the necessary level of stabilization.

(a) Critical, and hopefully also complex, functions might be put onto special-purpose chips which were then widely used. Such hardware embodiments would guarantee precise identity of structure over a wide range of applications, aid modularity, and reduce the run-time cost of using standardized rather than hand-tailored software components. Typical candidates for such embodiments might be chips to manage a

priority queue of processes, to handle interprocessor communications in a standard way, etc.

(b) Very-high-level library versions of common avionics functions can be developed and pre-certified. Note that formal correctness-proof methods will apply more easily to text of this kind than to more detail-ridden code. High-level, abstract program versions can serve as stable standards which can impart a useful degree of logical stability to production codes developed from them. As already noted, the availability of high-level libraries of standard algorithms can also facilitate the work of software audit teams.

(c) Efforts can be made to develop other kinds of strictly re-usable software modules or packages.

In particular, the following question could usefully be addressed: is it possible to develop any significant collection of software modules which can be used without change for a wide variety of aircraft?